Class #24: Garbage Collection; Methods/Functions/Subroutines

Programming Languages (CS 421/521): M. Allen, 03 Apr. 19

---

**Review: Mark-Sweep Garbage Collection**

- **free_list** contains all memory locations currently free
  - Each node in heap has a **mark bit**
  - Initially, all nodes are in the free_list, with mark bit = 0
- When heap overflow occurs, we make **two sweeps** over heap:
  1. Mark all nodes (directly or indirectly **accessible**) from active variables (i.e., in runtime stack) by setting mark bit = 1
  2. Sweep through the entire heap and return all the **unmarked** (mark bit = 0) nodes to free_list
- This process returns all **orphan** objects to free_list

---

**Heap after First Pass of Mark-Sweep**

- p and q are still active in run-time stack, so heap space to which they refer is marked with mark bit = 1
- Continues recursively for objects linked to p and q (internal variables, etc.), and for other live variables in run-time stack
- When the process is complete, orphaned objects still have mark bit = 0, as never reached during sweep

---

**Heap after Second Pass of Mark-Sweep**

- When first pass is complete, all memory nodes with mark bit = 0 are returned to the free_list for re-use
- After that, all mark bits are reset to 0 so that allocation and execution can continue
Garbage Collection in the JVM

- The JVM employs a slightly more complex version of the mark-sweep scheme
- The operation does not happen often (we hope)
  - As a program executes, newly created dynamic objects are given space on the heap
  - If the objects never fill the heap, no garbage is collected
  - Only attempted if the heap gets too full
  - This can be an intensive operation
- Programmers can request early collection, via a method call to `System.gc()`
  - The JVM may still decide not to do it at that point

Making Garbage Collection Possible

- A number of languages (Java, Scala, Go, C#, etc.) have managed to implement a robust mark-sweep model
- If a language is not strongly typed, generally not possible
  - E.g., in C there are cases where it is not possible to determine whether some value is actually a pointer to live memory (happens especially after casts on pointers)
  - In such cases, only a conservative garbage collector is safe
  - Such a collector may not be able to free up all memory that one in a language with stronger typing can
- Garbage collection cannot, also, deal with programmer-generated issues
  - If a programmer doesn’t free up references to objects that are no longer needed, there is generally little that the system itself can do

Methods/Functions: Basic Terminology

```java
int max(int a, int b) {
    int result;
    if (a > b) result = a;
    else result = b;
    return result;
}
```

- Some functions return values, while others do not
  - In Ada and Fortran, ones that do not return anything (procedures or subroutines) work differently
  - In C/C++/Java void and non-void types are often interchangeable

Classic Java error:

```java
String s = "hello";
s.toUpperCase();
System.out.println(s);
```

Method/Function Parameters

- When we define a method and then call it, the formal parameters are usually matched to the actual parameters (inputs) by position
  ```java
  public void setArea(int long, int wide) { ... }
  setArea(6, 50);
  ```
- Other languages, like Scala, OCaml, or Ada, allow us to label parameters to over-ride this behavior if we wish:
  ```scala
  def pow( base: Double, exp: Double ) = { ... }
  val x = pow(3, 2) // 9.0
  ```
  ```java
  def pow( base: Double, exp: Double ) = { ... }
  val x = pow(3, 2) // 9.0
  ```
  If we don’t use the labels, everything works the same
  If we use the labels, we can enter parameters in any order we choose
**Parameter Passing Styles**

- **Pass by Value**
  - When a function is called, information is passed between caller and called method.
  - **Direction of communication**
    1. **actual** parameters to **formal** parameters (in mode)
    2. **formal** parameters to **actual** parameters (out mode)
  - **When** does communication happen?
    a) at the start of the call
    b) throughout the call
    c) when the call ends

- **Pass by Reference**
  - An implementation of in-mode parameters.
  - The formal parameter is a local variable, given a value provided by caller.
  - The value of the actual argument is computed at time of call.
  - This value is used to initialize the corresponding formal parameter.
  - The only option in C, Java, OCaml, many others.
  - In C, can pass value of object address (reference pointer *) however.
  - In Java, OCaml complex (reference, array) type variables store address references by default.
  - In effect, the formal parameter becomes an alias for the actual parameter.
  - C++ allows us to designate some calls as by reference, otherwise uses values.

```c
void swap(int* a1, int* a2) {
    int temp;
    temp = a1;
    a1 = a2;
    a2 = temp;
}
```

**Passing Parameters**

- **Pass by value** (1,a)
  - Value of input parameter is accessible to **called method** at **start**
- **Pass by reference** (3,b)
  - Value of the parameter is shared between the caller **and** the method called.
- **Pass by value-result** (3,a,c)
  - Value of the parameter accessible to **called method** at **any time**
  - Changes to that value only known by the caller **after** the method ends.
- **Pass by name** (3,b)
  - It’s a lot more complicated…

**Pass by Value**

- **An implementation of in-mode parameters**
  - The formal parameter is a local variable, given a value provided by caller.
  - The value of the actual argument is computed at time of call.
  - This value is used to initialize the corresponding formal parameter.

**Pass by Reference**

- **An implementation of out-mode parameters**
  - The address of the argument is obtained at the time of call.
  - This address is given to the called method.
  - In effect, the formal parameter becomes an alias for the actual parameter.

- C++ allows us to designate some calls as by reference, otherwise uses values.

```c
void swap(int* a1, int* a2) {
    int temp;
    temp = *a1;
    *a1 = *a2;
    *a2 = temp;
}
```

In the early days, FORTRAN only allowed pass by reference, leading to many side-effects. A well-known bug allowed changes to “constants” like integer literals:

```c
void plus(int x) {
    x = x + 1;
}
```

```c
int i = 2;
plus(i);
```

```c
i == 3!```

---

**Wednesday, 3 Apr. 2019 Programming Languages (CS 421/521)**
Pass by Reference

- A main advantage of such a device is to allow large data structures to be input to methods, without copying the whole thing over again.
- This is why the feature was added to C++.
- Also why Java reference types are distinct.

Potential disadvantages arise from possible side-effects:

- Changes made by the method can affect the values of many variables outside the method itself.
- Such changes can be hard for programmers to keep track of.

Pass by Value-Result

- **Value-Result**: combines two modes to create in-out-mode parameters.
- Arguments are treated as local variables again.
- Copied **in** from calling environment at start of called method.
- Copied **out** to calling locale at end of method.
- Gets complicated when aliasing happens (two variables naming same thing).
- Not the same as pass by reference, where communication happens throughout the method call—instead, it only happens at start/end.

### Examples

**C++**

```cpp
void f(int &i1, int &i2) {
    i1 = i1 + 1;
    i2 = i2 + 1;
}
```

**Ada**

```ada
procedure f(i1, i2: in out Integer) is
    begin
        i1 := i1 + 1;
        i2 := i2 + 1;
    end f;
```

**Java**

```java
public void multiply(double x) {
    for (int i = 1; i <= 5; i++) {
        System.out.println(x * i);
    }
}
```

### Consider Java method:

```java
public void multiply(double x) {
    for (int i = 1; i <= 5; i++) {
        System.out.println(x * i);
    }
    multiply(Math.random());
}
```

**Pass by Name**

- Name of actual argument is **textually copied** to called method.
- Each use is evaluated at the time the code containing it is executed.
- Algol 60 utilized the method exclusively.
- Could cause surprising behavior, and abandoned.

### Consider Java method:

```java
public void multiply(double x)
{
    for (int i = 1; i <= 5; i++) {
        System.out.println(x * i);
    }
    multiply(Math.random());
}
```

**Pass by Name**

- What happens if we used pass by name instead of value?

```java
public void multiply(double x)
{
    for (int i = 1; i <= 5; i++) {
        System.out.println(x * i);
    }
}
```

**Actual input name** "Math.random()" is substituted for "x" everywhere.

### Every time it is used in the loop, we get a new random number!

```java
public void multiply(double Math.random())
{
    for (int i = 1; i <= 5; i++) {
        System.out.println(Math.random() * i);
    }
    multiply(Math.random());
}
```
This Week & Next

- **Topic:** Garbage collection; functions and subroutines
- **Reading:** Text, 8.5, 9.1–9.3
- **Meet:** Friday: No class
- **Homework 03:** due Friday, 12 April, 5:00 PM
- **Office Hours:** Wing 210
  - Thursday: 1:00 PM – 3:00 PM
  - Friday: No office hours this week
  - Back to normal next week